

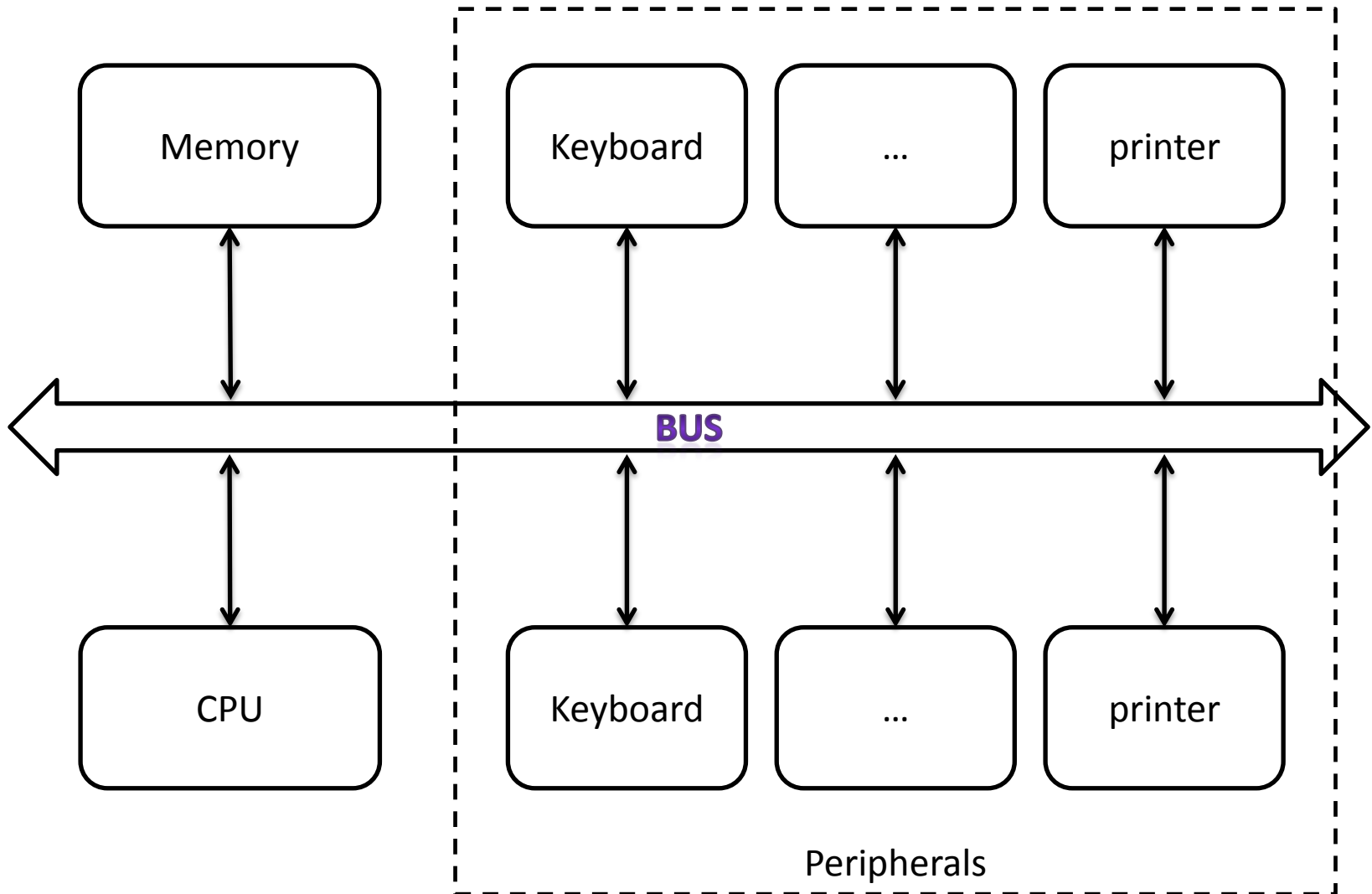
# Computer architecture

*A simplified model*

# Computers architecture

- One (or several) CPU(s)
- Main memory
- A set of devices (peripherals)
- Interrupts
- Direct memory access

# Computers architecture



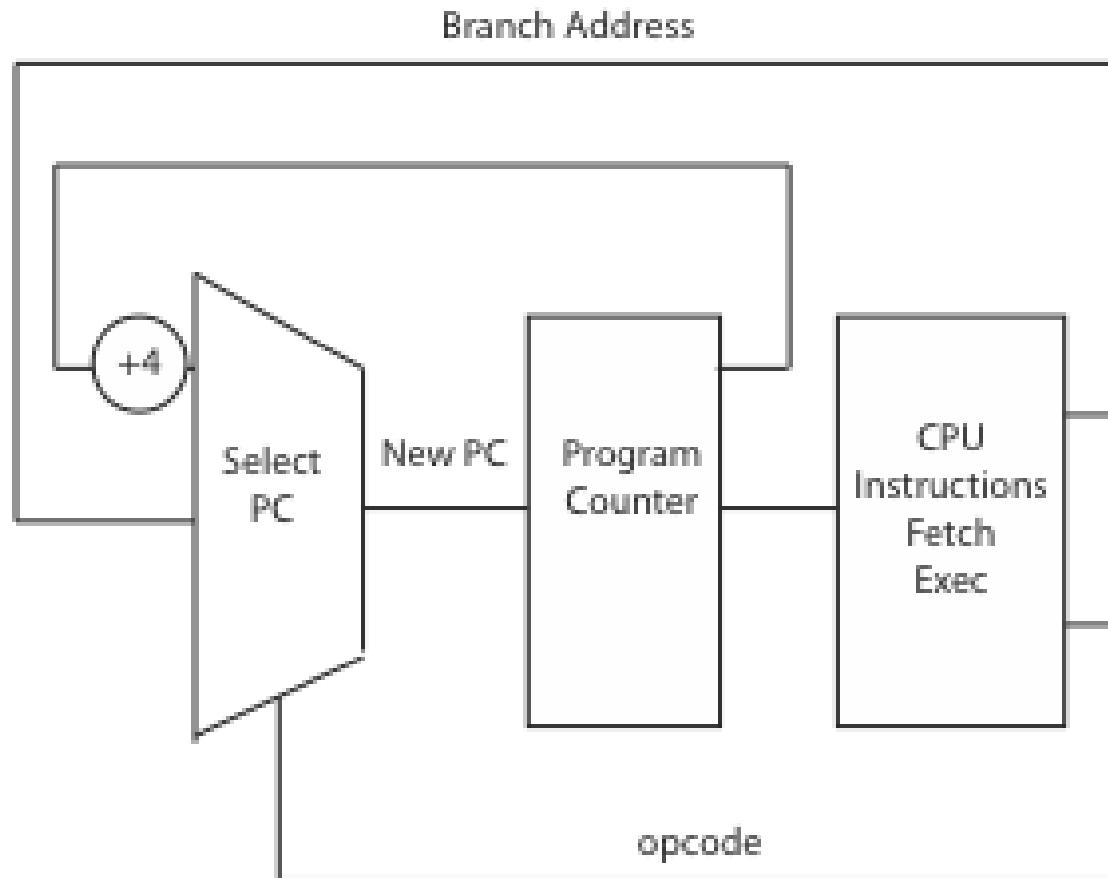
# The CPU

- General registers
- State and control registers
  - Program Counter (PC o IP)
  - Stack Pointer (SP)
  - Program Status register (PS)

# Fetch-execution cycle

- If there are pending interrupts and the interrupts are enabled
    - Manages the interrupt
  - Else
    - Loads the instruction at address PC
    - Executes the instruction
    - $PC = PC + 4$  (\*)
- (\*) assumes that the instruction occupies 4 bytes

# A Model of a CPU



# The program status register

- Condition code
  - Keeps the status of the last instruction (divide by zero, overflow, carry etc.)
- CPU mode
  - User mode VS kernel mode
- Interrupt enable bit

# The Kernel Abstraction



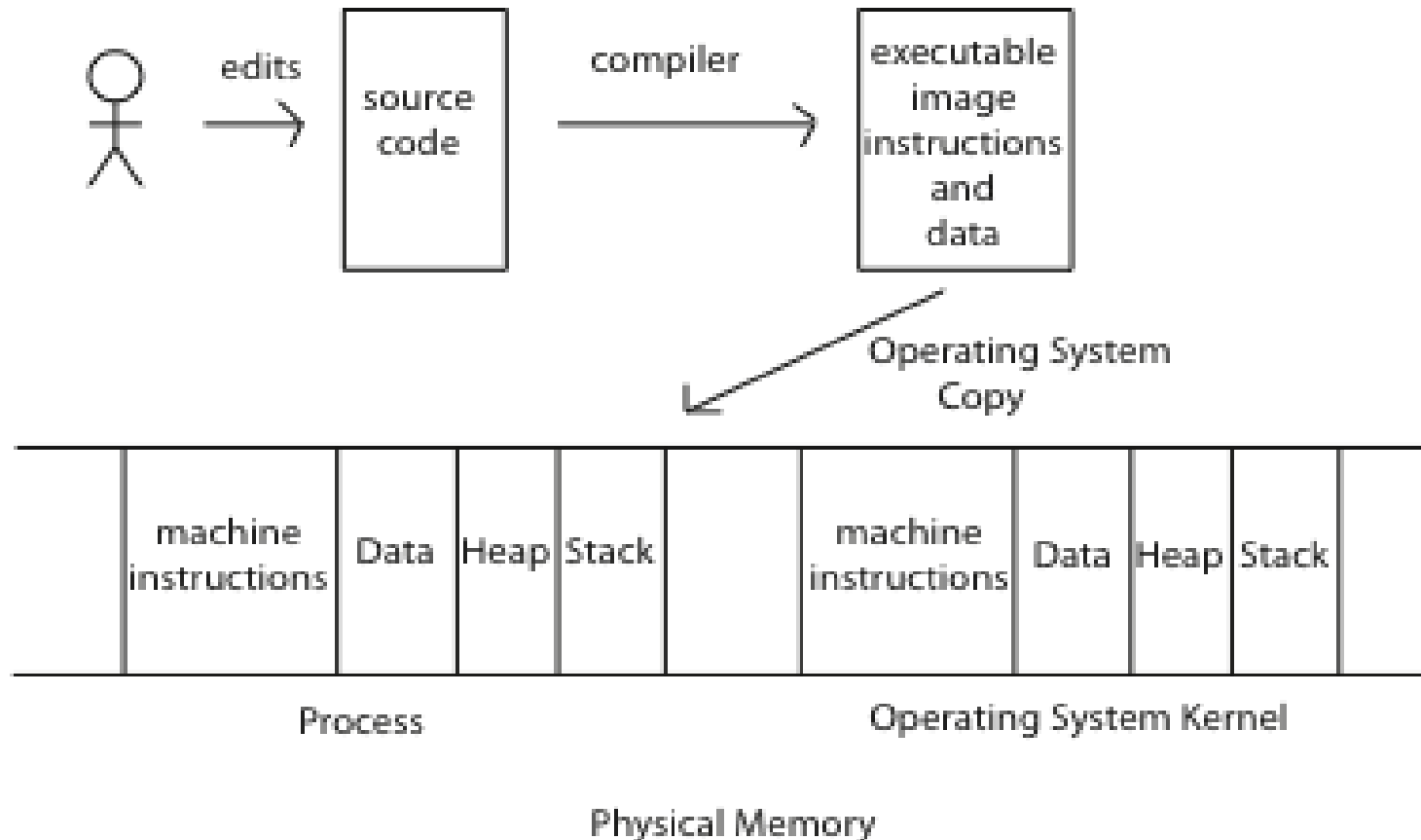
# Challenge: Protection

- How do we execute code with restricted privileges?
  - Either because the code is buggy or if it might be malicious
- Some examples:
  - A script running in a web browser
  - A program you just downloaded off the Internet
  - A program you just wrote that you haven't tested yet

# Main Points

- Process concept
  - A process is an OS abstraction for executing a program with limited privileges
- Dual-mode operation: user vs. kernel
  - Kernel-mode: execute with complete privileges
  - User-mode: execute with fewer privileges
- Safe control transfer
  - How do we switch from one mode to the other?

# Process Concept



# Process Concept

- Process: an instance of a program, running with limited rights
  - Process control block (PCB): the data structure the OS uses to keep track of a process
  - Process Table: contains all PCBs
  - Two parts to a process:
    - Thread: a sequence of instructions within a process
      - Potentially many threads per process (for now 1:1)
      - Thread aka lightweight process
    - Address space: set of rights of a process
      - Memory that the process can access
      - Other permissions the process has (e.g., which procedure calls it can make, what files it can access)

# Program and process

- Program: sequence of instructions
- Process: a set of activities executed on a set of CPUs
- Several processes can be activated on the same program
  - The processes execute the same code
  - Each process executes the program on different data and/or in different times

# Process Control Block

- Process name
  - Can be the index of the PCB in the process table
- Pointers to process threads
- Assigned memory
- Other resources
  - Files, devices, etc...

# Thought Experiment

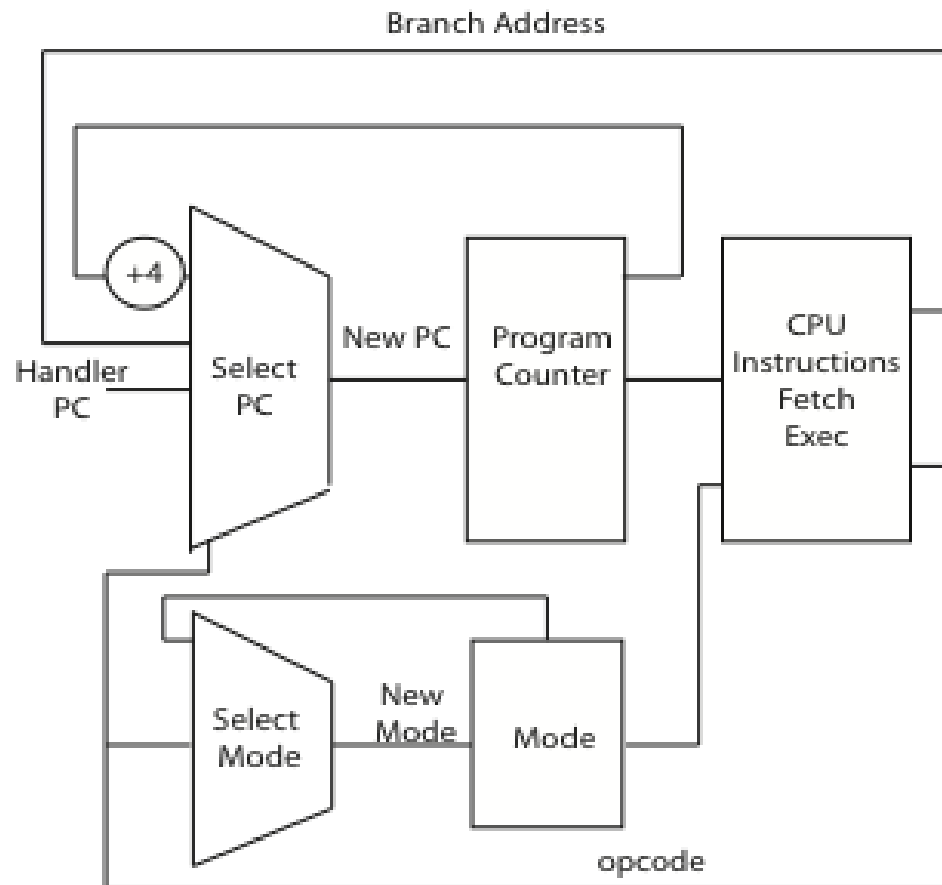
- How can we implement execution with limited privilege?
  - Execute each program instruction in a simulator
  - If the instruction is permitted, do the instruction
  - Otherwise, stop the process
  - Basic model in Javascript, ...
- How do we go faster?
  - Run the unprivileged code directly on the CPU?

# Hardware Support: Dual-Mode Operation

- Kernel mode
  - Execution with the full privileges of the hardware
  - Read/write to any memory, access any I/O device, read/write any disk sector, send/read any packet
- User mode
  - Limited privileges
  - Only those granted by the operating system kernel
- On the x86, mode stored in EFLAGS register



# A CPU with Dual-Mode Operation



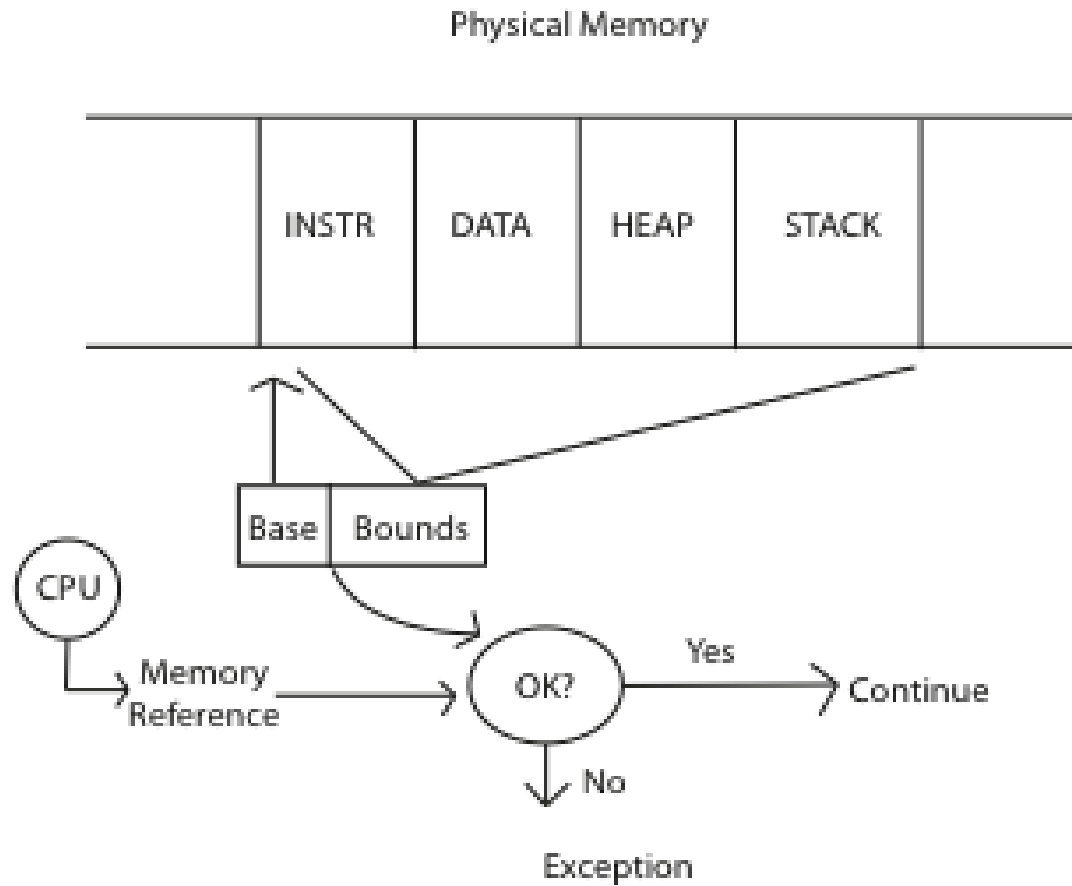
# Hardware Support: Dual-Mode Operation

- Privileged instructions
  - Available to kernel
  - Not available to user code
- Limits on memory accesses
  - To prevent user code from overwriting the kernel
- Timer
  - To regain control from a user program in a loop
- Safe way to switch from user mode to kernel mode, and vice versa

# Privileged instructions

- Examples?
- What should happen if a user program attempts to execute a privileged instruction?

# Memory Protection

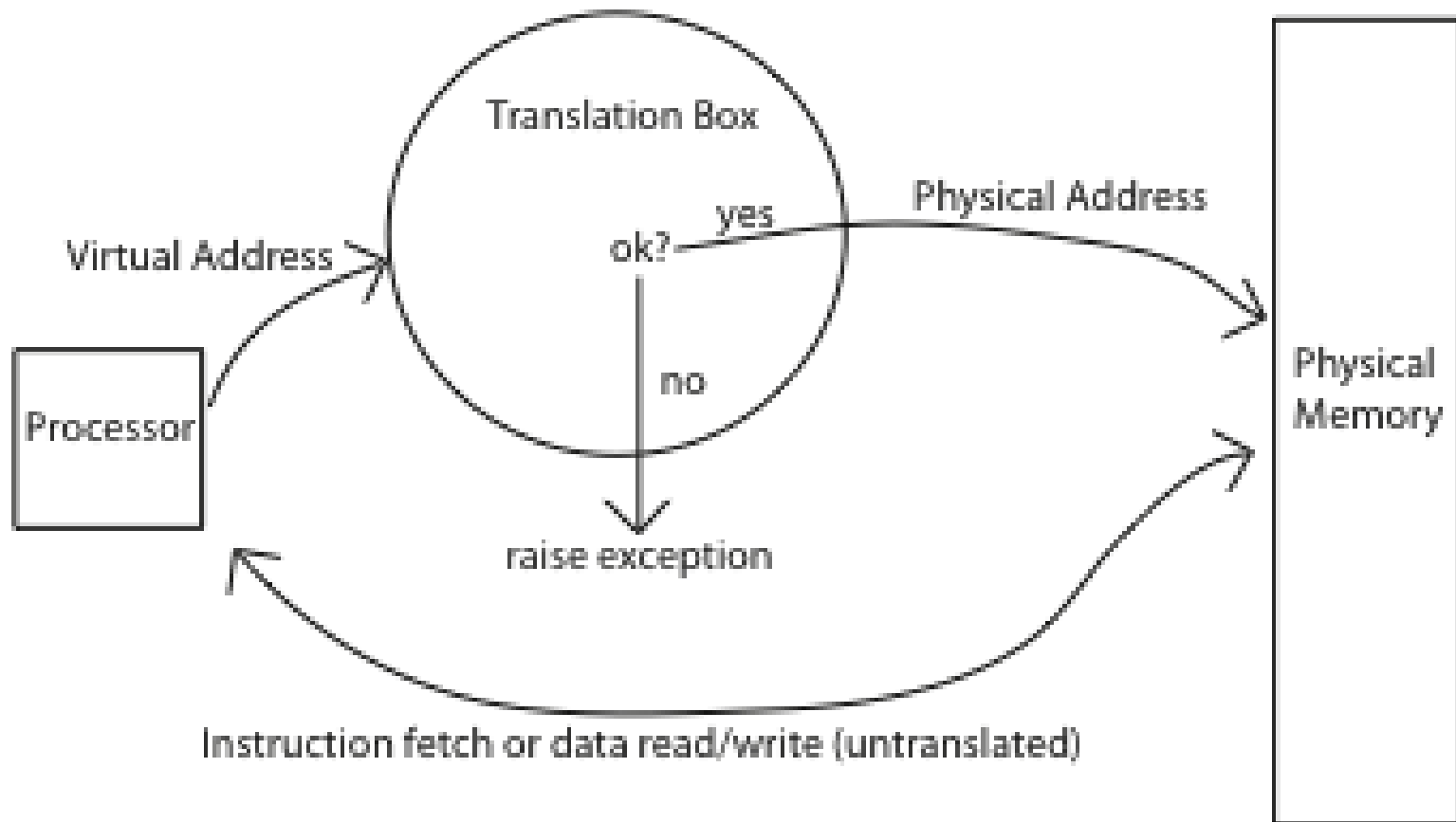


# Towards Virtual Addresses

- Problems with base and bounds?

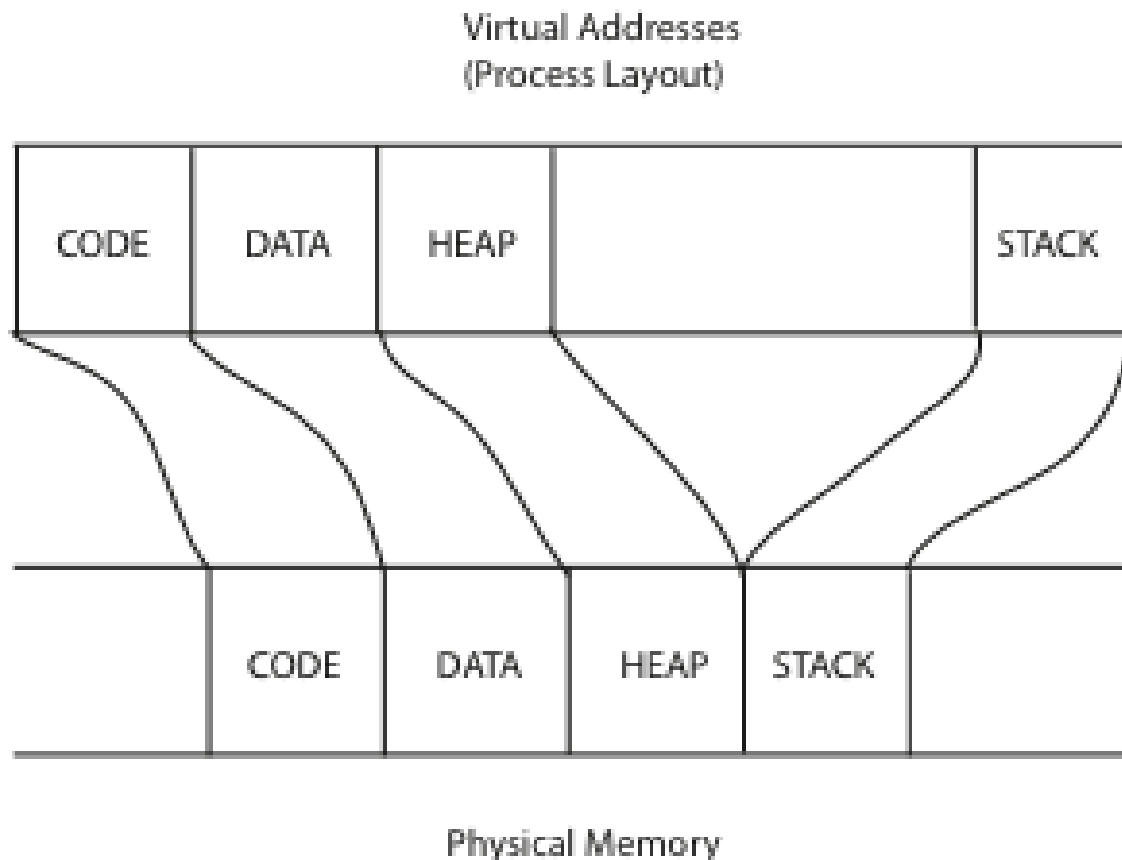
# Virtual Addresses

- Translation done in hardware, using a table
- Table set up by operating system kernel



# Virtual Address Layout

- Plus shared code segments, dynamically linked libraries, memory mapped files, ...



# Example: Corrected (What Does this Do?)

```
int staticVar = 0;    // a static variable
main() {
    int localVar = 0; // a procedure local variable

    staticVar += 1; localVar += 1;

    sleep(10); // sleep causes the program to wait for x seconds
    printf ("static address: %x, value: %d\n", &staticVar, staticVar);
    printf ("procedure local address: %x, value: %d\n", &localVar, localVar);
}
```

Produces:

```
static address: 5328, value: 1
procedure local address: fffffffe2, value: 1
```



# Hardware Timer

- Hardware device that periodically interrupts the processor
  - Returns control to the kernel timer interrupt handler
  - Interrupt frequency set by the kernel
    - Not by user code!
  - Interrupts can be temporarily deferred
    - Not by user code!
    - Crucial for implementing mutual exclusion

# Question

- For a “Hello world” program, the kernel must copy the string from the user program memory into the screen memory. Why must the screen’s buffer memory be protected?

# Question

- Suppose we had a perfect object-oriented language and compiler, so that only an object's methods could access the internal data inside an object. If the operating system only ran programs written in that language, would it still need hardware memory address protection?

# Mode Switch

- From user-mode to kernel
  - Interrupts
    - Triggered by timer and I/O devices
  - Exceptions
    - Triggered by unexpected program behavior
    - Or malicious behavior!
  - System calls (aka protected procedure call)
    - Request by program for kernel to do some operation on its behalf
    - Only limited # of very carefully coded entry points

# Mode Switch

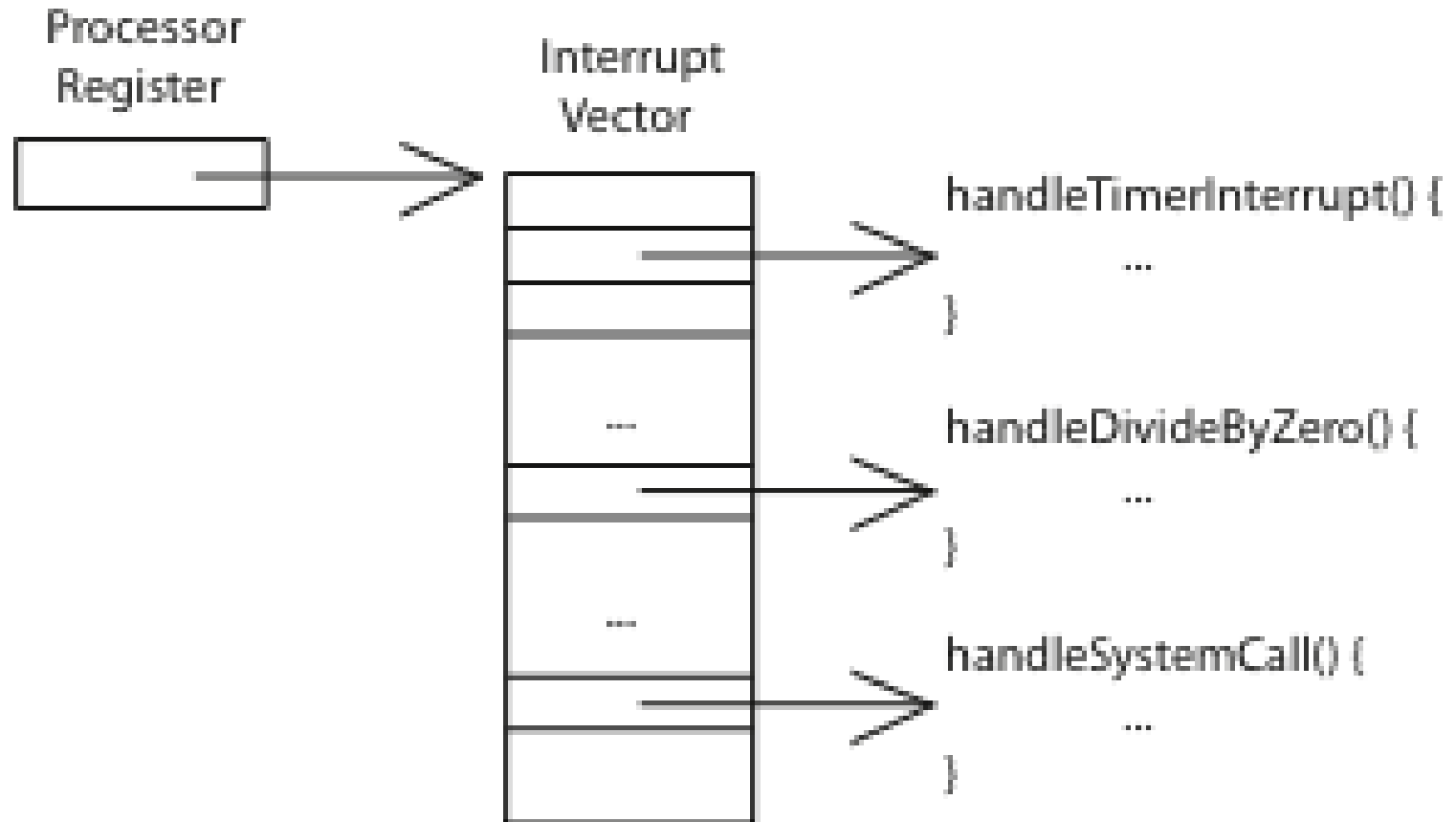
- From kernel-mode to user
  - New process/new thread start
    - Jump to first instruction in program/thread
  - Return from interrupt, exception, system call
    - Resume suspended execution
  - Process/thread context switch
    - Resume some other process
  - User-level upcall
    - Asynchronous notification to user program

# How do we take interrupts safely?

- Interrupt vector
  - Limited number of entry points into kernel
- Kernel interrupt stack
  - Handler works regardless of state of user code
- Interrupt masking
  - Handler is non-blocking
- Atomic transfer of control
  - Single instruction to change:
    - Program counter
    - Stack pointer
    - Memory protection
    - Kernel/user mode
- Transparent restartable execution
  - User program does not know interrupt occurred

# Interrupt Vector

- Table set up by OS kernel; pointers to code to run on different events

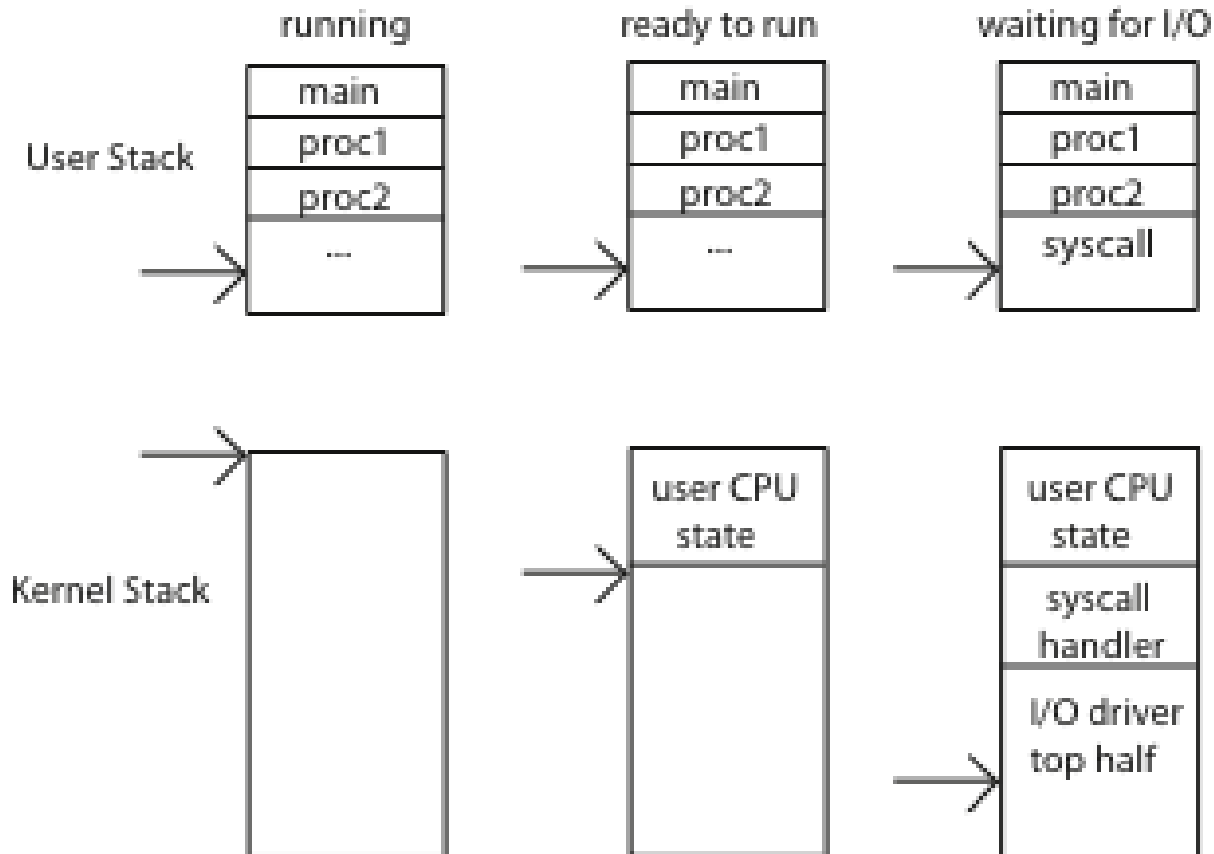


# Interrupt Stack

- Per-processor, located in kernel (not user) memory
  - Usually a thread has both: kernel and user stack
- Why can't interrupt handler run on the stack of the interrupted user process?



# Interrupt Stack



# Interrupt Masking

- Interrupt handler runs with interrupts off
  - Reenabled when interrupt completes
- OS kernel can also turn interrupts off
  - Eg., when determining the next process/thread to run
  - If defer interrupts too long, can drop I/O events
  - On x86
    - CLI: disable interrupts
    - STI: enable interrupts
    - Only applies to the current CPU
- Cf. implementing synchronization, chapter 5

# Interrupt Handlers

- Non-blocking, run to completion
  - Minimum necessary to allow device to take next interrupt
  - Any waiting must be limited duration
  - Wake up other threads to do any real work
    - Pintos: semaphore\_up
- Rest of device driver runs as a kernel thread
  - Queues work for interrupt handler
  - (Sometimes) wait for interrupt to occur

# Atomic Mode Transfer

- On interrupt (x86)
  - Save current stack pointer
  - Save current program counter
  - Save current processor status word (condition codes)
  - Switch to kernel stack; put SP, PC, PSW on stack
  - Switch to kernel mode
  - Vector through interrupt table
  - Interrupt handler saves registers it might clobber

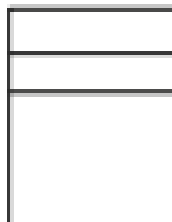
# Before

User-level  
Process

code:

```
foo () {  
  while(...) {  
    x = x+1;  
    y = y-2;  
  }  
}
```

stack:



Registers

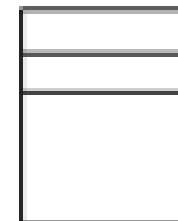


Kernel

code:

```
handler() {  
  pusha  
  --  
}
```

Exception  
Stack



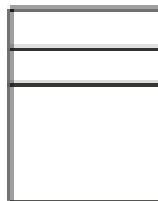
# During

User-level  
Process

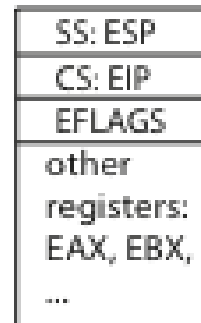
code:

```
foo () {  
    while(...) {  
        x = x+1;  
        y = y-2;  
    }  
}
```

stack:



Registers

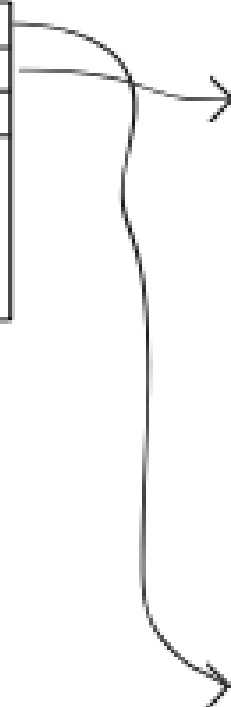


Kernel

code:

```
handler() {  
    pusha  
    --  
}
```

Exception  
Stack



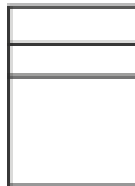
# After

User-level  
Process

code:

```
foo () {  
  while(...) {  
    x = x+1;  
    y = y-2;  
  }  
}
```

stack:



Registers



Kernel

code:

```
handler() {  
  pusha  
  ...  
}
```

Exception  
Stack



# At end of handler

- Handler restores saved registers
- Atomically return to interrupted process/thread
  - Restore program counter
  - Restore program stack
  - Restore processor status word/condition codes
  - Switch to user mode
- IRET instruction



# Interrupt management

## a simple example

Initial state: interrupt '500' occurs when executing instruction A000

Registri nella CPU

PC	A000
PS	PSW P
SP	FFFF
R1	AAAA
R2	BBBB
...	

Memoria

programma P

...	...
A000	istr. 1
A004	istr. 2
A008	istr. 3
A016	istr. 4
A020	istr. 5
...	...

Stack di P

...	...
FFF0	
FFF3	
FFF7	
FFFB	
FFFF	...

stack nel nucleo

...	...
2996	
2997	
2998	
2999	
3000	

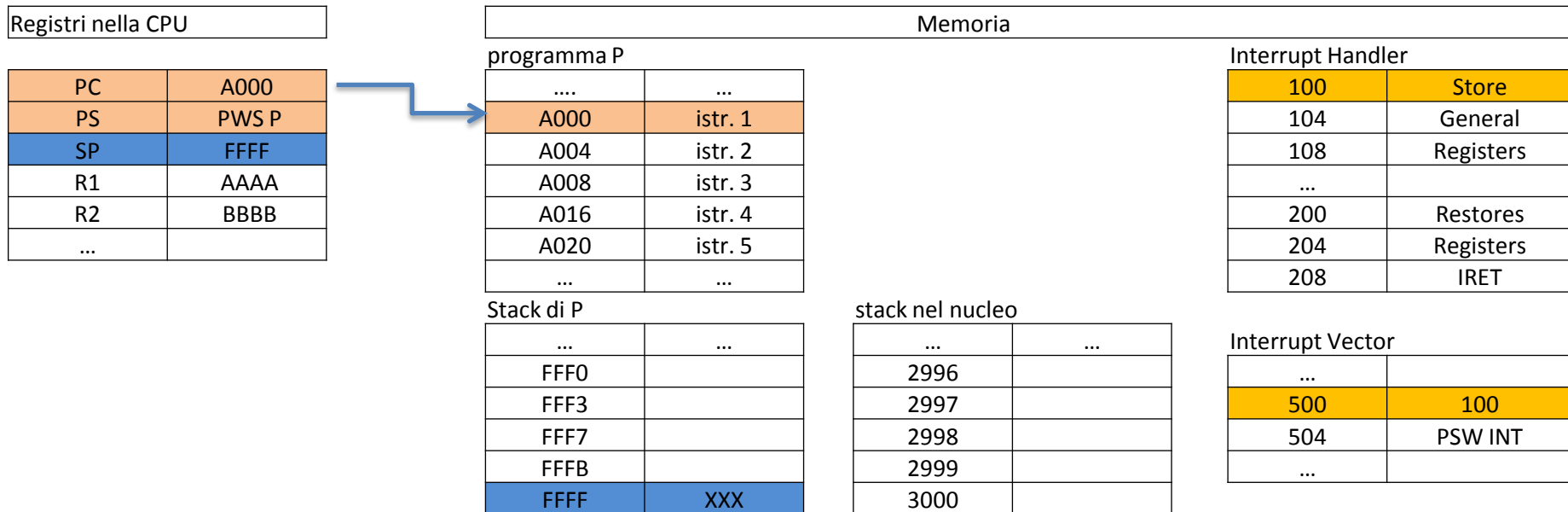
Interrupt Handler

100	Store
104	General
108	Registers
...	
200	Restores
204	Registers
208	IRET

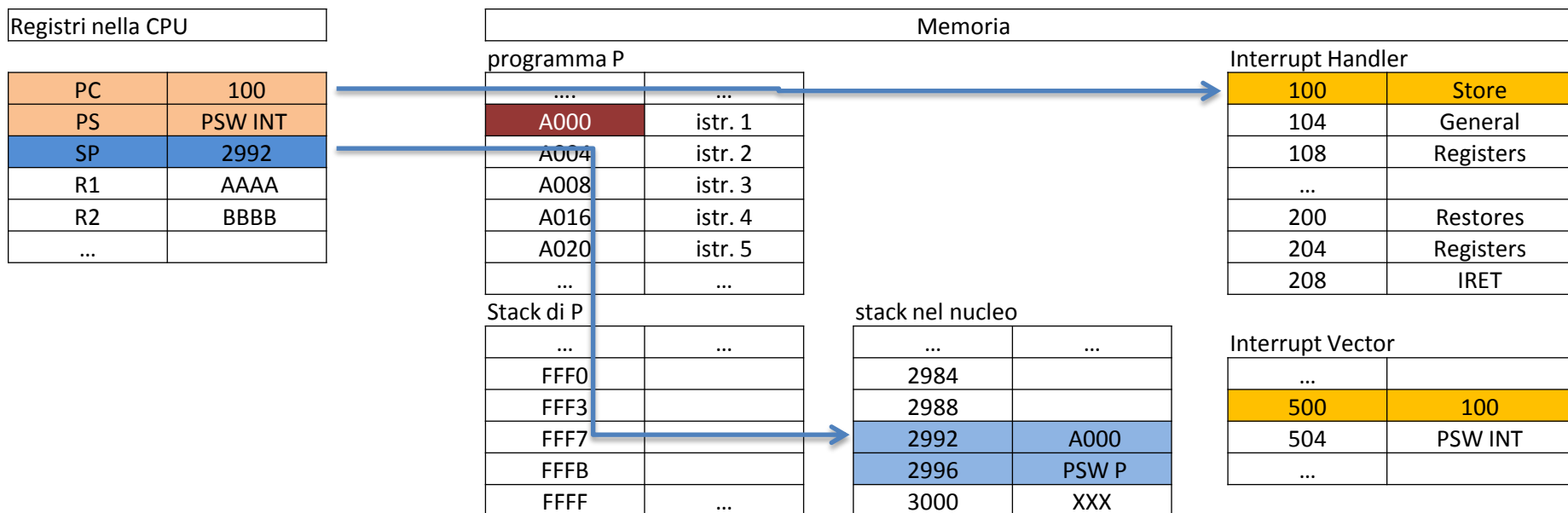
Interrupt Vector

...	
500	100
504	PSW INT
...	

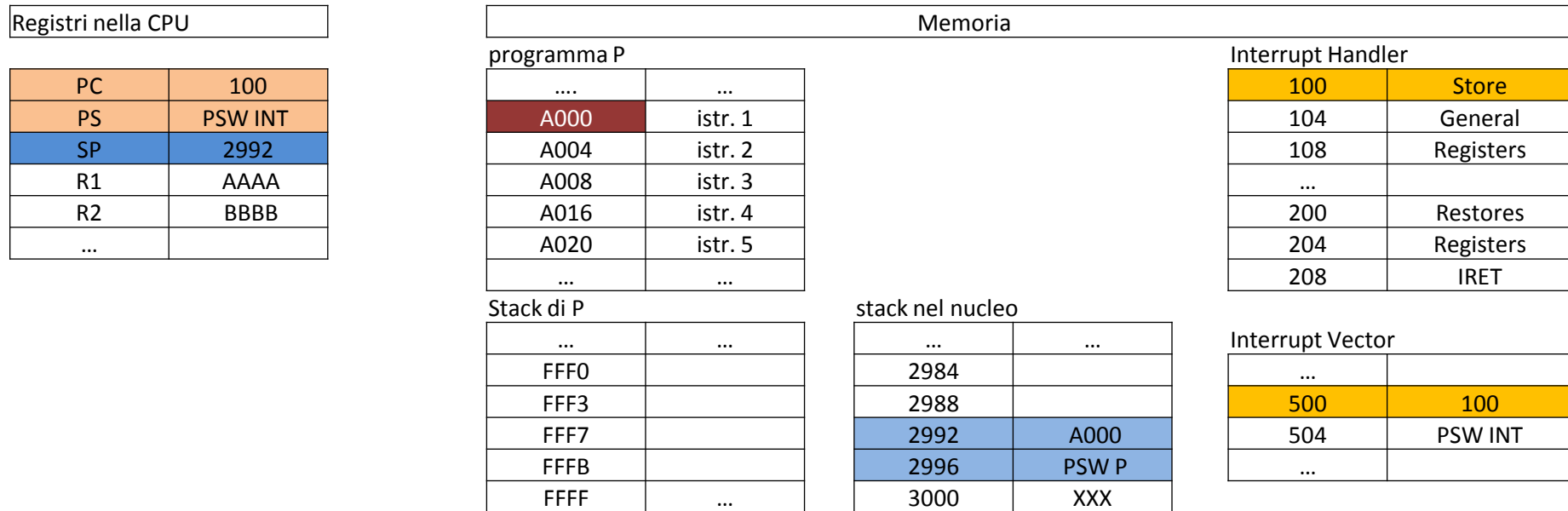
# 1) Initial state



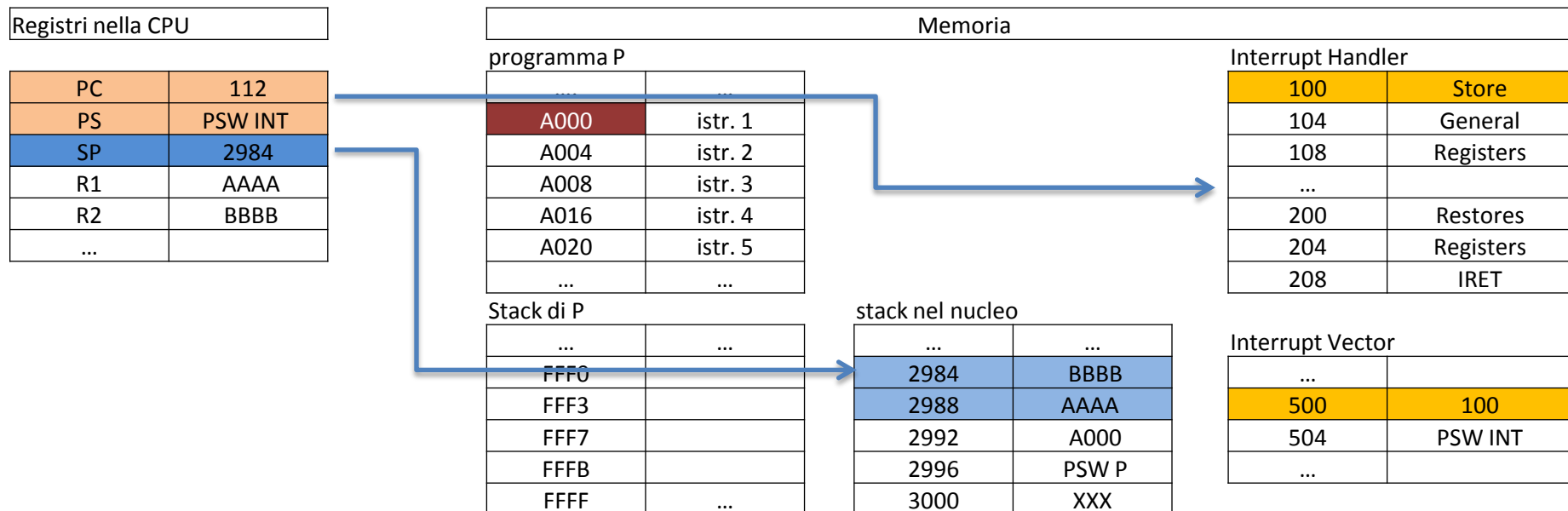
# 2) Interrupt recognized after instruction A000



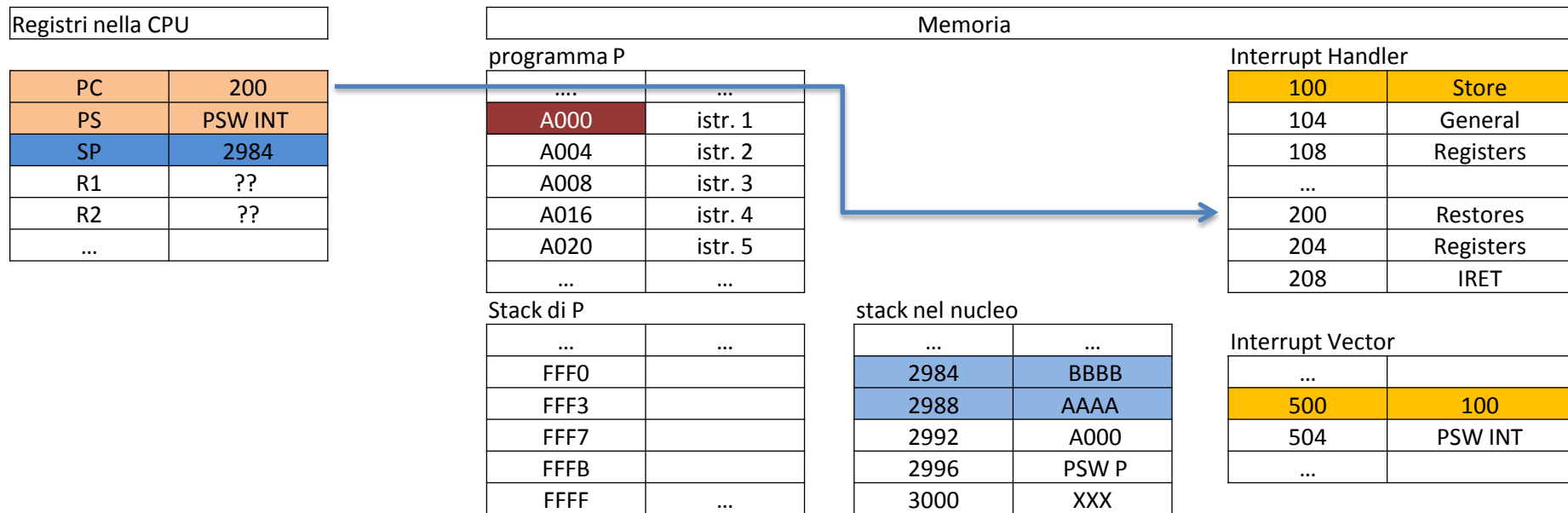
## 2) Interrupt recognized after instruction A000



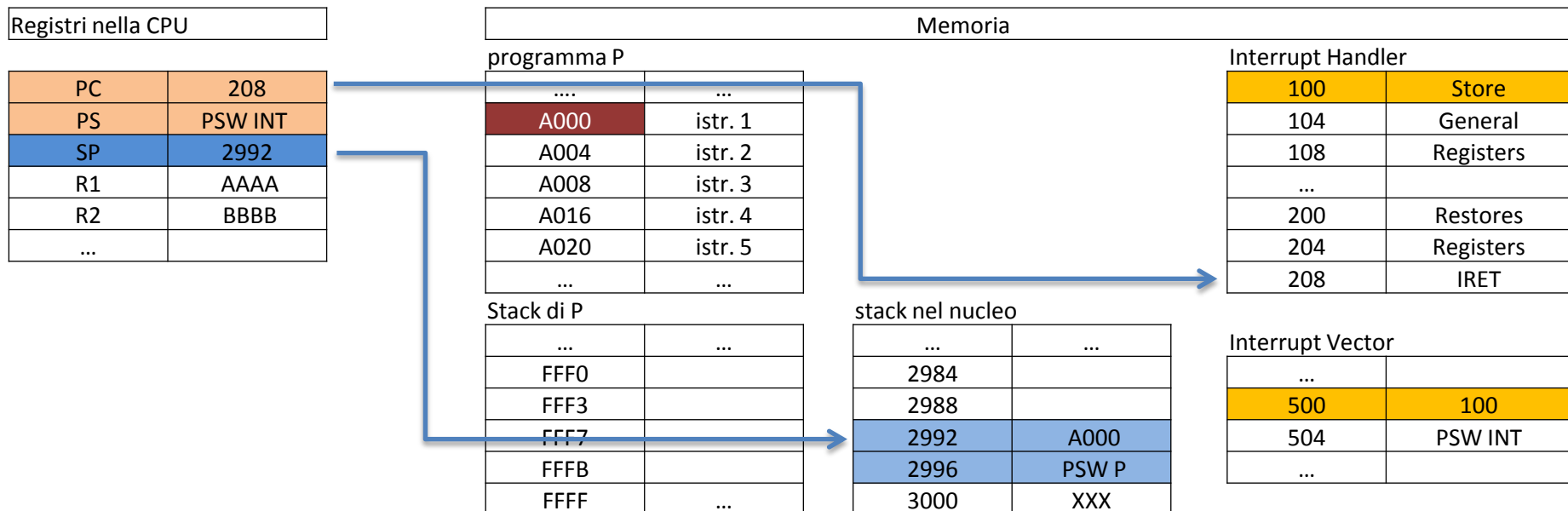
## 3) Stores general registers



## 4) Executes interrupt handler



## 5) Restores general registers



## 5) Restores general registers

Registri nella CPU

PC	208
PS	PSW INT
SP	2992
R1	AAAA
R2	BBBB
...	

Memoria

programma P

...	...
A000	istr. 1
A004	istr. 2
A008	istr. 3
A016	istr. 4
A020	istr. 5
...	...

Interrupt Handler

100	Store
104	General
108	Registers
...	
200	Restores
204	Registers
208	IRET

Stack di P

...	...
FFF0	
FFF3	
FFF7	
FFFB	
FFFF	...

stack nel nucleo

...	...
2984	
2988	
2992	A000
2996	PSW P
3000	XXX

Interrupt Vector

...	
500	100
504	PSW INT
...	

## 6) Executes IRET

Registri nella CPU

PC	A004
PS	PSW IP
SP	2992
R1	AAAA
R2	BBBB
...	



Memoria

programma P

...	...
A000	istr. 1
A004	istr. 2
A008	istr. 3
A016	istr. 4
A020	istr. 5
...	...

Interrupt Handler

100	Store
104	General
108	Registers
...	
200	Restores
204	Registers
208	IRET

Stack di P

...	...
FFF0	
FFF3	
FFF7	
FFFB	
FFFF	...

stack nel nucleo

...	...
2984	
2988	
2992	
2996	
3000	XXX

Interrupt Vector

...	
500	100
504	PSW INT
...	

# System Calls

User Program

```
main () {  
    ...  
    syscall(arg1, arg2);  
    ...  
}
```



User Stub

```
syscall (arg1, arg2) {  
    trap  
    return  
}
```

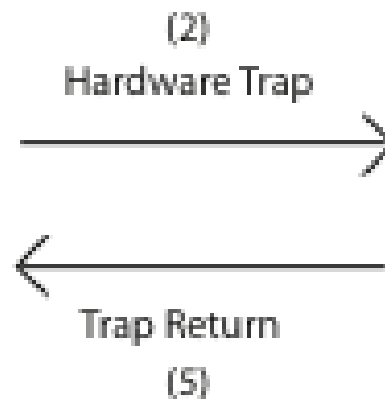
Kernel

```
syscall(arg1, arg2) {  
  
    do operation  
  
}
```



Kernel Stub

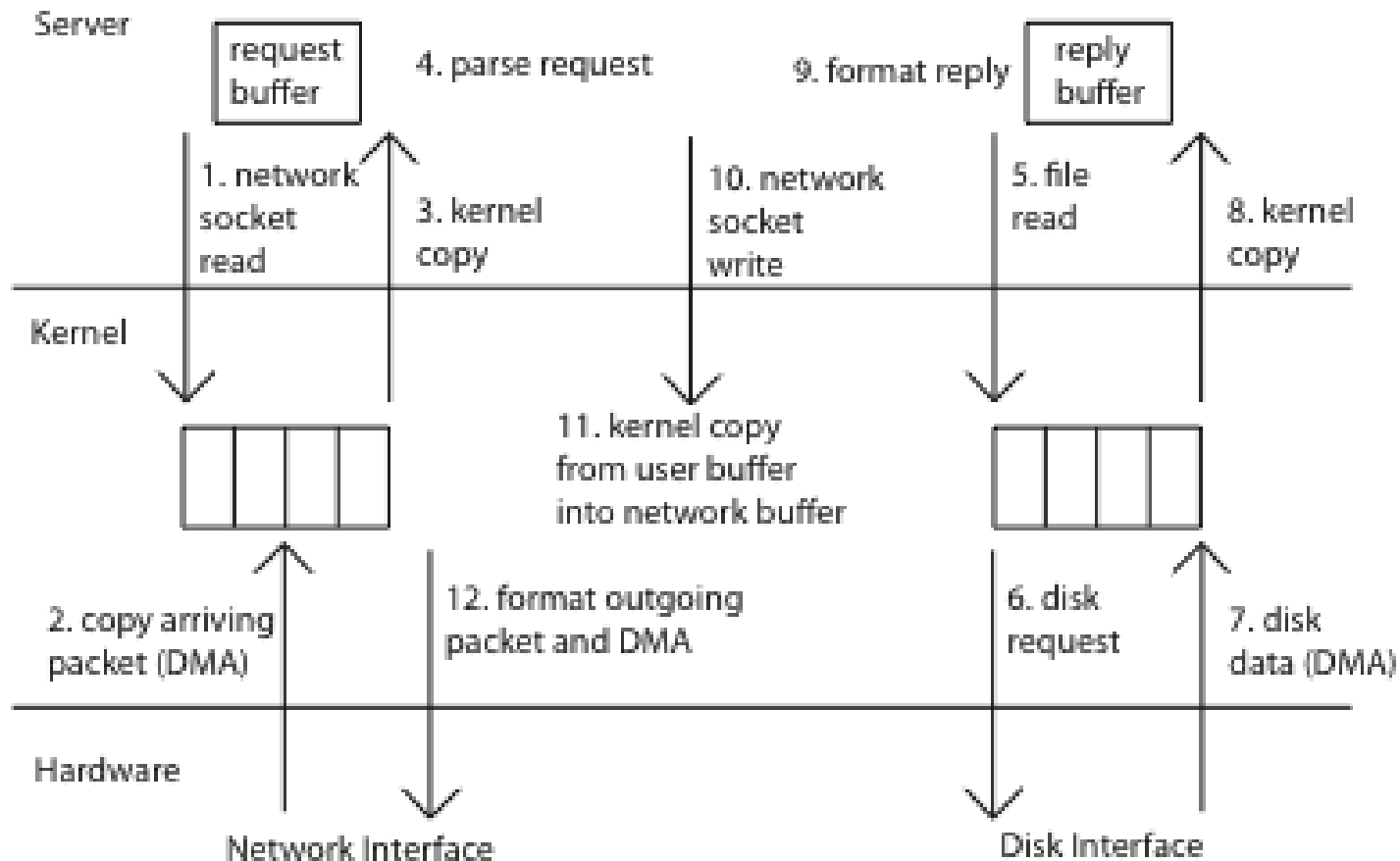
```
handler() {  
    copy arguments  
    from user memory  
    check arguments  
    syscall(arg1, arg2);  
    copy return value  
    into user memory  
    return  
}
```



# Kernel System Call Handler

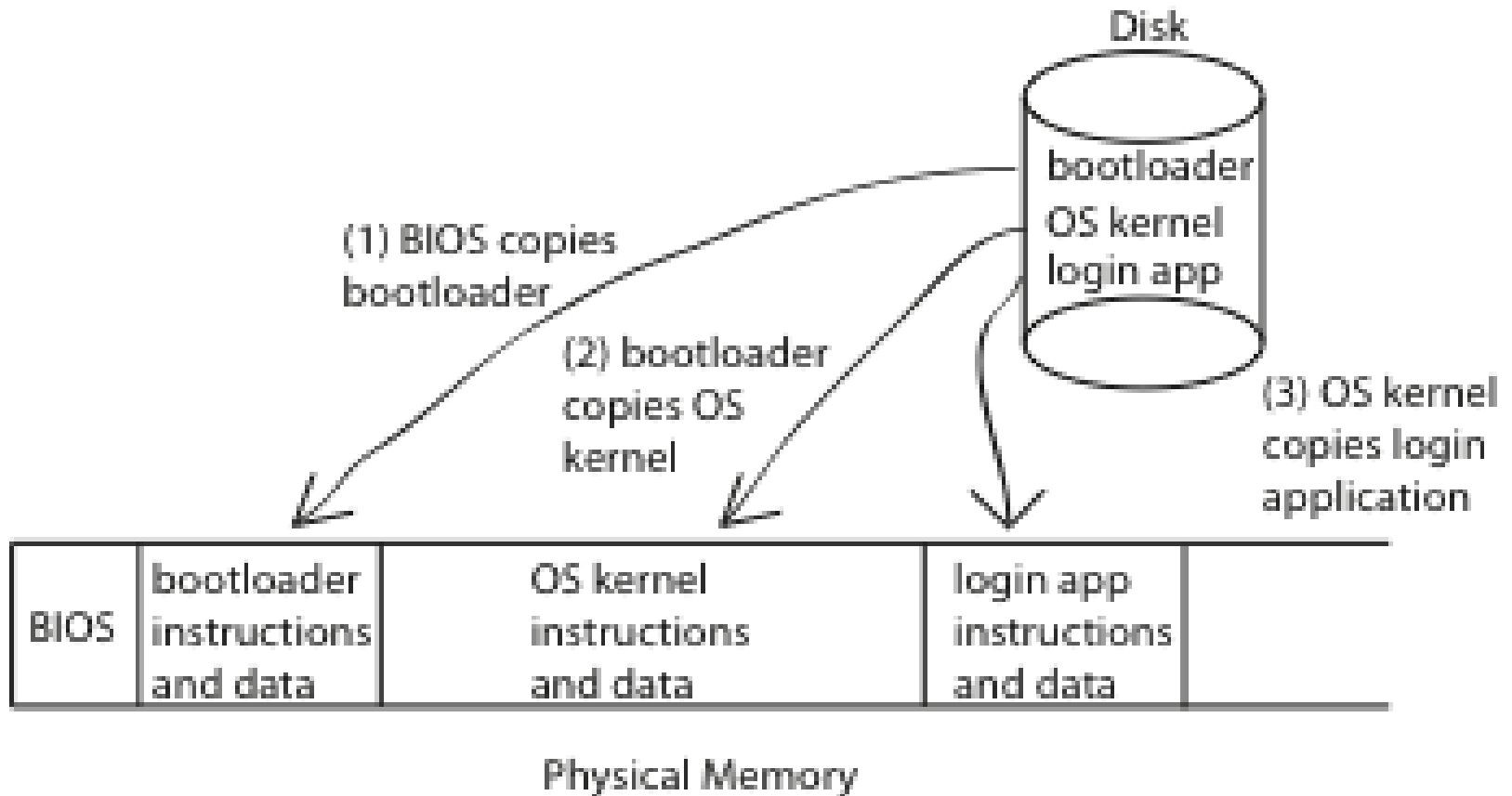
- Locate arguments
  - In registers or on user(!) stack
- Copy arguments
  - From user memory into kernel memory
  - Protect kernel from malicious code evading checks
- Validate arguments
  - Protect kernel from errors in user code
- Copy results back
  - into user memory

# Web Server Example

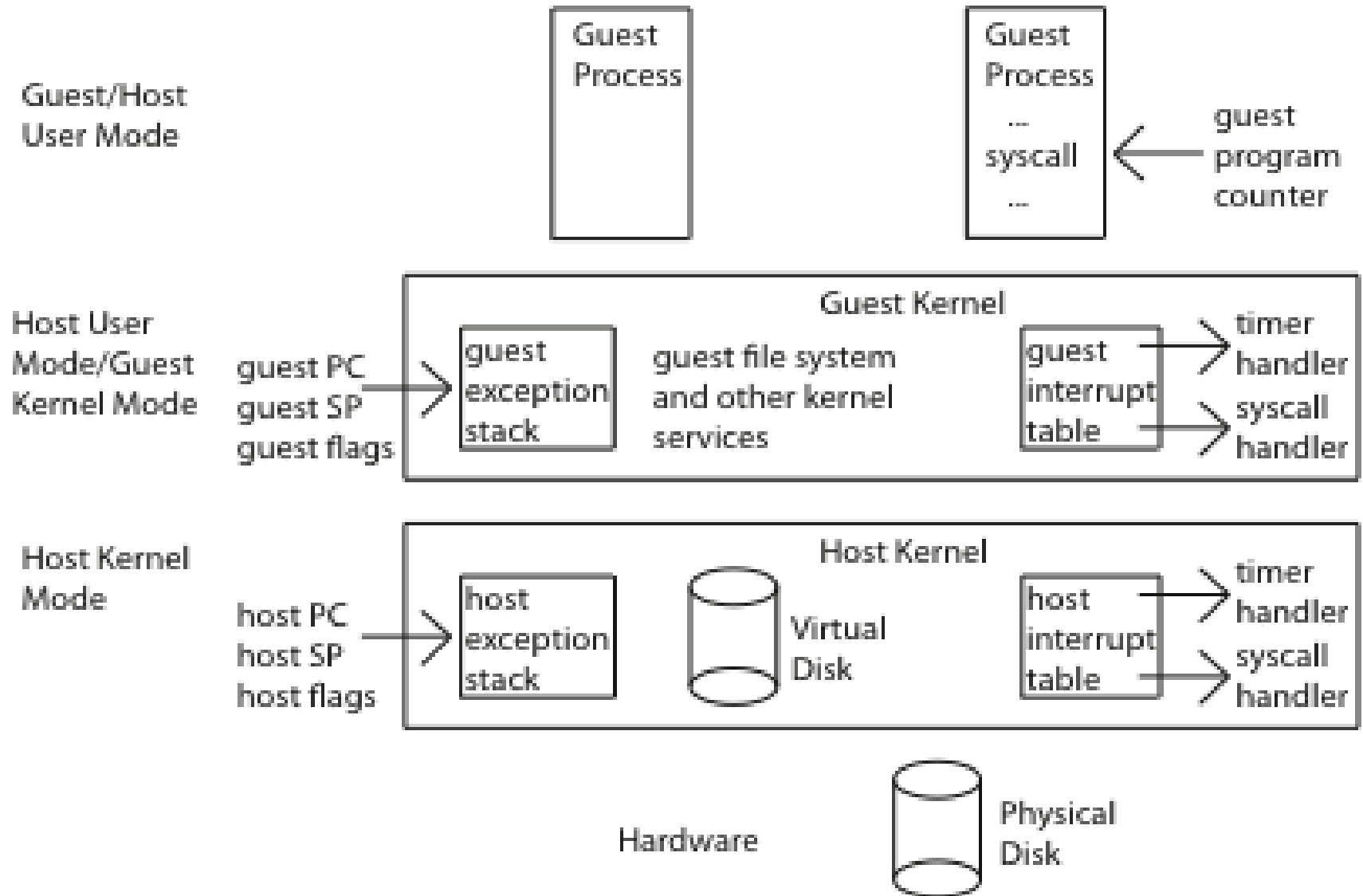




# Booting



# Virtual Machine



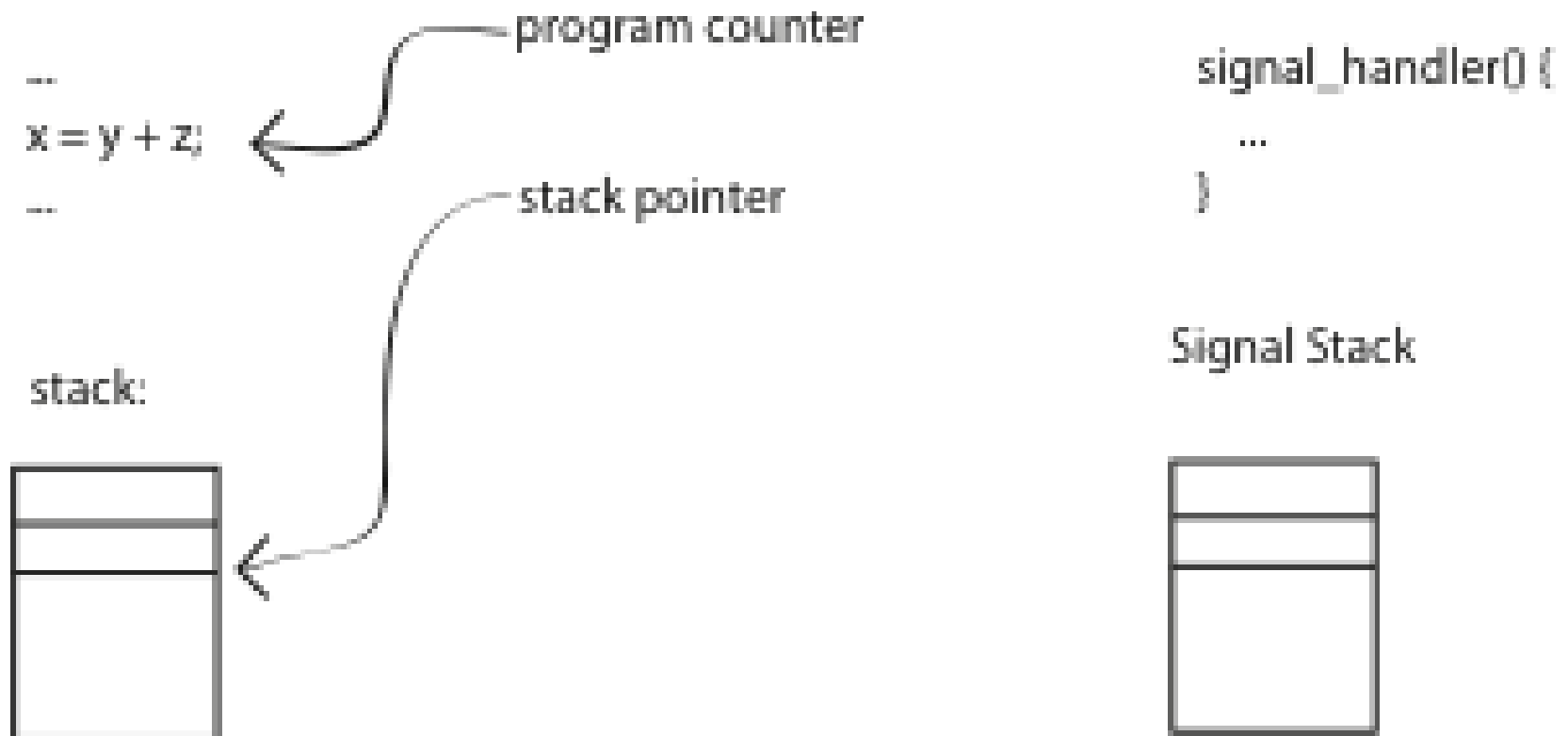
# User-Level Virtual Machine

- How does VM Player work?
  - Runs as a user-level application
  - How does it catch privileged instructions, interrupts, device I/O, ...
- Installs kernel driver, transparent to host kernel
  - Requires administrator privileges!
  - Modifies interrupt table to redirect to kernel VM code
  - If interrupt is for VM, upcall
  - If interrupt is for another process, reinstalls interrupt table and resumes kernel

# Upcall: User-level interrupt

- AKA UNIX signal
  - Notify user process of event that needs to be handled right away
    - Time-slice for user-level thread manager
    - Interrupt delivery for VM player
- Direct analogue of kernel interrupts
  - Signal handlers – fixed entry points
  - Separate signal stack
  - Automatic save/restore registers – transparent resume
  - Signal masking: signals disabled while in signal handler

# Upcall: Before



# Upcall: After

